ICOT Technical Report: TR-005

TR-005

ESP
as a Preliminary Kernel Language of Fifth Generation Computers

by

Takashi Chikayama

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191 ~ 5
Telex ICOT J32964

Institute for New Generation Computer Technology

ESP
as a Preliminary Kernel Language
of Fifth Generation Computers

Takashi CHIKAYAMA

Institute for New Generation Computer Technology
Research Center

Mita Kokusai Building, 21F.
4-28, Mita 1-chome, Minato-ku, Tokyo 108

ABSTRACT

In the first three-year development stage of the fifth generation computer systems project, a series of high-performance personal computers called sequential inference machines are being developed at ICOT Research Center. The machines have a high-level machine language called KL0, which is a PROLOG-based logic language with various extensions. In the software development of the sequential inference machines, ESP, a software-supported yet higher level language compiled into KL0, is used instead of directly using KL0. This paper describes the design the language system of sequential inference machines. Description will be centralized on ESP with an overview of KL0.

1. Introduction

In the first three-year development stage of the fifth generation computer systems project, a series of high-performance personal computers called sequential inference machines are being developed at ICOT Research Center, the first version of which is called PSI . Various application software systems as well as the operating system and programming systems for itself will be developed on these inference machines in this first stage, and will be used as development tools for more advanced systems in the succeeding stages of the project.

The machines have a high-level machine language called KL0 (Kernel Language version-0), which is a logic language based on PROLOG with various extensions and certain features omitted. Omitted features are supplemented by the software-supported language ESP, which is compiled into KL0. All the software on the machine, including even the most basic hardware-oriented portions of the operating system will be described in ESP.

Main features of the ESP language other than those of PROLOG are:

- o Module Structure,
- o Macro Expansion, and
- o Global Data Definition.

The relationship of ESP and KL0 parallels the relationship of so-called system implementation languages and machine languages on conventional computers.

This paper first describes the design principles of the total language system of the sequential inference machines including ESP and KL0, then the design of KL0 briefly, and then

the design of various features of ESP.

2. Design Principles

Basic design objectives of the language system of sequential inference machines are the following:

- o Sufficient Power required for describing the objective software systems,
- o Brevity and Consistency for ease of both implementing and learning the language, and
- o Reasonable Efficiency of implementation.

Not all the features of the language system have to be attained by the hardware-supported language KL0. It would be all right if they could be attained by the total system consisting of KL0 and ESP. Therefore, features rather simple but requiring efficiency are included in the machine language KL0, while features rather complex or requiring flexibility are implemented by the software system ESP.

3. An Overview of KL0

KL0 is the machine language of the sequential inference machines. It is based on PROLOG with various extensions and certain features omitted. The main extensions to PROLOG are:

- o Extended Control Structure,
- o Multiple Processes,
- o Operations with Side-Effects, and
- o Hardware-Oriented Operations,

while significant omitted features are:

- o Database Management, and

o Name Table Management.

3.1 Data Types

KL0 has five basic data types, namely, symbols, integer and real numbers, strings and vectors.

Symbols are mainly used for representing symbolic literal atoms of PROLOG. Symbols in KL0 have basically no association with character strings used in textual representation of programs and data, nor with predicate definitions associated with a symbol when used as a predicate name. When such attributes are required to be attached to symbols, they should be implemented by some software systems. KL0 is a more primitive language than having such attribute mechanisms built-in. It merely provides random access data structures and a standard hashing function for supporting hash table implementation of such attribute definitions. Symbols of KL0 exist only for the sake of its identity.

Integer and real numbers are provided for efficiently executing arithmetical operations. Arithmetical operations in KL0 are not bi-directional: Addition and subtraction should be effected by individual operations. The hardware provided numerical types are only of fixed bit width. Arbitrary length integer numbers (bignums), arbitrary precision real numbers, and probably rational numbers are to be implemented using exception handlers: When operands that are not numerical objects are passed to built-in arithmetical predicates (machine instructions, in conventional sense), an exception is raised; its handler can examine the arguments, do appropriate operations if arguments are

something expected by the exception handler, call the error handler if not, and then resume normal execution.

Strings are a one-dimensional array of small positive integer data. Its element size, which constrains the value range of elements, varies from one string to another. Strings with 1-bit elements are used for representing bit arrays, such as memory-mapped graphic display images; those with 8-bit elements are for ASCII character strings; those with 16-bit elements are for character strings containing 16-bit Japanese character codes.

Vectors are a one dimensional array of arbitrary KL0 objects. A vector can contain arbitrarily many elements as long as the storage capacity allows. Indices for vector elements are from zero up to the length of the vector minus one. Thus, the first element of a vector has its index value zero. It is called the principal element of the vector. Elements of a vector can be accessed by its index value in a constant time regardless of the size of the vector.

As vectors are the only composite structure in KL0 language which allows arbitrary type elements, they are used for representing almost everything. Thus, compound terms of PROLOG-like languages such as "f(X,Y)" are, of course, represented by a vector, with its principal element being the principal functor and the rest of the elements being the arguments. In PSI, when a vector is not very long, its length is stored in the tag attached to the pointer word minimizing the required storage.

Vectors are also used for representing LISP-like linked list structures. A list is a sequence of vectors linked by their

first elements. Thus, when a vector is treated as a binary list cell, its first element is interpreted as its cdr and the second as its car. Using this scheme, list structures can be accessed more efficiently, in terms of number of memory references required, than when represented using compound terms with cons as their functors as seen in several PROLOG implementations. Storage required does not differ much if a structure sharing implementation should be used. There can be no ambiguity between lists and usual compound terms if the terminator of lists should be restricted to a certain object, say, a symbol nil, and that terminator should never be used as a functor of compound terms: The principal element of a compound term is always a non-nil symbol, while that of a list cell is the list terminator nil, another list cell, or uninstantiated.

There are several other data types not mentioned here for internal system management purposes, which may not be very useful for ordinary users.

3.2 Allocation of Data

There are two kinds of vectors in KL0, stack vectors and heap vectors. Though both are treated to be of the same type and may be unifiable, there are certain differences between them.

Stack vectors are allocated on a stack implicitly while unification process so that the allocated area can be freed on backtrack. They may contain uninstantiated logical variables, values of which are determined through unification and retracted by backtracking. Structure sharing method is adopted for stack vectors in our first machine PSI.

Heap vectors are allocated by explicit operations in a heap area which is not freed on backtrack. They are ground literals without logical variables, but assignment to them as side effects is provided by built-in predicates. The effect of assignment is not retracted by backtracking.

Heap vectors are mainly used for implementing databases where side-effects are required for efficient implementation. Another important purpose of allocating vectors in heap areas is for communication between processes. As each process has its own stack area which grows and shrinks independently, stack areas should never be accessed from other processes to avoid the problem of dangling pointers. Thus, inter-process communication has to be effected using heap vectors.

Symbols and numbers do not require storage allocation because they are embedded in a pointer word. Strings are always allocated in heap areas. The reason of representing strings by a separate type, rather than using vectors, is nothing but to minimize the required storage: it is no use allocating stack strings whose elements can be variables and thus require one word each.

3.3 Control Structure

The control structure of KL0 is basically that of PROLOG, depth-first search of an AND-OR tree. In addition to this, KL0 provides several more flexible control primitives.

A mechanism is introduced for delaying execution of a program segment until the instantiation of a specified variable, which can be found in PROLOG-II as freeze.

An extended version of cut operation called remote cut is introduced. For remote cuts, the point up to which alternative branches are pruned can be arbitrarily specified by marking the point with a label using a certain built-in predicate and specify that label as an argument of the remote cut operation. This point may not necessarily be an ancestor node of the node where the cut operation is carried out. Any point whose execution is sequentially prior to the cut operation (and not backtracked) can be specified. This feature, along with failure, can be used to implement a dynamically scoped non-local exit mechanism similar to that provided by catch and throw control structures seen in several LISP systems.

On the other hand, the semantics of usual cut operations is slightly changed so that it prunes alternatives up to the most recent ancestor OR-node, in stead of up to the most recent ancestor predicate call. By this change, OR-nodes appearing in a clause body can always be translated into a call of a separate predicate. This consistency makes definition of macros for customized control structures easier. The original somewhat awkward semantics seems to be required for controlling the scope of cut operations, which is more generally solved by remote cuts in KL0.

ESP provides another catch and throw control structure which throws success up to a labeled point, rather than throwing failure. The labeled AND-node completes successfully, with possibly remaining intervening AND branches ignored temporarily. When backtracking came back to the node which succeeded by this success throw, the throwing of success is assumed to be a failure

and the execution process continues as usual.

Exception handlers can be defined dynamically while execution of programs for both predefined and user-defined exceptions. The handler is called in the context of the exception. After fixing the problem, it can continue the program execution by either succeeding or failing the predicate call where the exception took place. Exception handlers are intended not only to handle program errors, but also to handle cases which is not a bug but cannot be handled by the hardware and it is inefficient to always watch for such cases by the software. One typical example is implementation of more general arithmetics mentioned above.

Predicate definitions are represented by code type objects. Basically, a predicate call is represented by a pointer to this code object with its arguments following it. However, this pointer is allowed to designate the code object via arbitrarily many indirection words. This mechanism is convenient for implementing module structure of ESP. By providing a vector of entries for each module and restricting module entry accesses to those through that vector, representation of module entries can be revised by rewriting only this entry vector without changing the codes which use the module.

4. Notation of ESP

4.1 Names

Names are used to denote various entities of ESP programs. A name is either a short name such as "a", or a full name

prefixed with the module name where the entity is defined and a colon like "m:a". Entries defined in the same module, or those defined in modules declared to be visible by a visibility declaration, can be accessed by their short names. Other entries must be named using their full names.

4.2 Structures

A vector is denoted by enclosing list of its elements by braces. For example, a vector with 3 elements "a", "b" and "c" is denoted by "{a, b, c}". This is the most basic notation for structured data of ESP. Other notations are merely a short-hand of this vector notation. The first (index 0) element of a non-null vector is called the principal element of the vector. When the principal element of a vector is a symbol, it is sometimes called the principal functor of the vector.

There are three kinds of strings in ESP. They have 1-bit, 8-bit and 16-bit integer elements and are called bit strings, byte strings and double-byte strings respectively. Byte strings and double-byte strings have a character string format to denote their values, like "ESP". Strings of all three types can be denoted in a form similar to vectors: The string type name followed by a list of component integer values enclosed in braces, like "byte{69,83,30}".

A compound terms such as "f(a,b)" is used to denote a vector whose principal element is a symbol. The length of the vector is the number of arguments in the argument list of the compound term plus one. Its principal (index 0) element is the functor symbol and following elements (index 1 and up) are the arguments. For

example, a compound term of the form "f(a,b)" is interpreted as a vector "{f, a, b}".

List structures, which actually consist of binary vectors, can be denoted using list notations. A list notation of the form "[X | Y]" is a short-hand of "{Y, X}" (note that elements are reversed). A list notation of the form "[X, ...]" is a short-hand of "{ [...] | X }". A null list denotes a symbol "[]". Thus, a list notation "[a, b, c]" is a short-hand for "{ { {[], c}, b}, a}".

4.3 Operators

Operator applications also are used to denote vectors with a symbolic principal element. A prefix or a postfix operator application denotes a vector with two elements whose principal (index 0) element is the operator itself, and second (index 1) element is the operand. An infix operator application denotes a vector with three elements whose principal element is the operator itself, the second element, the left hand side operand, and third element, the right hand side operand. The order of operator applications is determined by their precedence relation. Operators can be defined along with their precedence using operator definitions given in modules.

In ESP, precedence relation between operators is determined based on precedence relations between two operators explicitly given in operator definitions (figure 1), rather than by giving precedence value to each operator. This scheme is adopted because it expresses the programmer's intention in a more direct manner.

Precedence relation is affected by whether one operator appears left or right to the other operator. For example, the infix operator "+" precedes "-" when "-" appears to the right of the appearance of "+", but is preceded by "-" in a reverse situation. This makes "A + B - C" interpreted as "(A + B) - C" and "A - B + C" as "(A - B) + C".

Suppose two operators X and Y appears textually in this order. In this case, the problem is whether X precedes right Y, or Y precedes left X, or neither. In such a case the precedence of operators X and Y is determined according to the following rules:

(1) Precedence Definitions

If there is a precedence definition for X in the operator definition of X saying that "X precedes right Y", then X precedes Y. Otherwise, if there is a precedence definition for Y saying "Y is preceded by left X", then X precedes right Y.

(2) Transitivity

If X precedes right Z and Z precedes right Y, then X precedes right Y.

Operators defined in a module other than the current one is assumed to be defined when the name of the defining module is given in the visibility declaration. To avoid cases where two operators precede each other, a precedence definition resulting circular precedence is erroneous: Defining an operator to be preceding another operator which, according to already given precedence definitions and the transitivity rule, precedes the original operator, is checked out by the compiler.

5. Macro

Macros are used for writing meta-programs, which specify that programs with so and so structure should be interpreted as such and such programs. Even somewhat complicated macros can easily be defined in forms of ESP programs fully utilizing the pattern matching and logical inference capability of the logic programming language. Macro expansion of ESP does not only replace the macro invocation with the expanded result, but also can augment the program part including the invocation with certain runtime conditions.

5.1 Definition

The syntax of macro definition is shown in figure 2. Basically a macro definition consists of the invocation pattern and the expanded pattern. Both are usual terms and may include logical variables. When one appears in both, they refer to the same entity. A sub-term in an invocation of this macro, which appears at the position corresponding to a variable in the invocation pattern, will be included in the expanded result at the position where that same variable appears in the expanded pattern.

The optional runtime conditions can be specified to augment the program part including the macro invocation so as to be executed when the expanded result is executed. They are meant to be used for either generating or checking the value or some component of the value of the expanded result. The way they are expanded will be described in following sections.

The expansion condition also is a condition associated with

the macro expansion. It is executed when a macro invocation is expanded, rather than being included in the expansion and executed in runtime. The execution process of the expansion condition is similar to that of the body of a clause. The expansion condition can be used for dual purposes: to determine whether the macro expansion should be effected or not, and to determine the value of some of the uninstantiated variables included in the expanded pattern, depending on the arguments given in the invocation pattern.

5.2 Invocation and Expansion

A macro invocation is any term which can be unified with the invocation pattern of a certain macro definition and the execution of its expansion condition succeeds.

When a pattern looking like a macro invocation pattern should be interpreted as it is, rather than being expanded, it should be quoted using a prefix operator "`" (back quote). The quoted term is not expanded and taken as it is except that the quoting operator "`" is eliminated. This quoting is effective only for the top-level of the pattern. Thus, the quoted term has macro invocations as its sub-terms, they will be expanded as usual.

Invocations appearing in a goal in the body of a clause and those in the head of a clause are expanded in slightly different manners. Invocations appearing in a clause body, including cases where goals themselves are a macro invocation, are expanded as follows:

(1) The invocation is unified with the invocation pattern of

the macro definition,

(2) The expansion condition of the macro definition, if any, is executed in the same way as the body of a clause,

(3) The goal including the invocation is replaced by a logical conjunction of following three in this order:

1) Generator,

2) Original goal with the concerning invocation substituted by the expansion,

3) Checker.

The order of the runtime conditions and the expanded goal is thus defined so as to be able to generate values of the expanded part before the goal is executed, and check the result of the goal after its execution.

The third step of the expansion process is different for invocations appearing in the head of a clause:

(3) The clause head including the invocation is replaced by the original head with concerning invocation substituted by the expanded pattern. The body is replaced by a logical conjunction of following three in this order:

1) Checker,

2) Original body,

3) Generator.

By this ordering, the value of the argument is checked at the beginning and generated at the end of the execution of the body.

Figure 3 shows a macro definition which allows functional notation of the adding operator "+", and examples of invocations. This is a typical example of using a macro for generating values.

Figure 4 shows a macro definition of the operator "I" and example usages. This is a typical example of using a macro for checking values. Readers may notice the difference in ordering of the expanded result.

Expanded result of a macro may include another macro invocation which in turn will be expanded, including the cases where the expanded pattern itself is another macro invocation. However, macro expansions are tried only in a top-down manner: Once a term is examined and found not being a macro invocation, it will never be treated as a macro invocation even if later macro expansions of sub-terms of this term have made the parent term unifiable with a certain macro invocation pattern.

6. Global Data

Global data definitions are used for defining a heap data structure which can be accessed through its name rather than by passing it as an argument to a predicate. This global data belongs to the module where it is defined, and can be initiated when the module is loaded by the initiator (figure 5).

When a module containing global data definitions is loaded, the initiator of each global data definition is executed in the same way as the body of a clause is executed. After the execution of the initiator, the global data (a term) is copied to a heap area if it is a compound term. By this copying global data becomes a heap object which can be assigned values. The term defining the global data may initially contain variables but they all must be instantiated during the execution of the initiator. From then on, that data can be accessed using the

name of the global data.

Global data is intended to be used mainly for two purposes. One is to eliminate the number of arguments of predicate calls. As, in PROLOG, the scope of variables is limited to a single clause, all the data required for computation must be passed as arguments. This certainly makes programs easy to read, for the reader can concentrate on data which is passed as argument, as long as the number of arguments is not too many. However, for considerably complicated programs, number of required arguments becomes too much for the reader to comprehend them all. For example, standard input and output streams (or value holders for them) must be passed as arguments in strictly pure coding style. Programmers are tempted to always include such commonly used data in the argument list, regardless of whether they are used or not. Such coding style merely makes the programs complicated. If they are represented as global data, programs which use only standard I/O streams do not always carry them all around among their arguments.

Another purpose of using global data is to record computed result of backtracked branches of execution. Programs requiring considerable efficiency, like the operating system kernel, are desired to be programmed in a style where repetitions are implemented by failure, because, using this style, memory area allocated on the stack can be reclaimed very efficiently. This optimization can automatically done by the machine optimizer when repetition are described using deterministic tail recursive calls with no arguments. In that case, the result of each repetition can only be recorded in global data as side-effects.

7. Modules

An ESP program is a collection of modules. A module is a collection of definitions which are closely related in a certain sense. A module definition consists of several declarations describing relations with other modules and definitions of various kinds (figure 6).

Modules may define predicates, macros, global data, operators and symbols. Among them, only those explicitly declared to be the entries of the module, can be used from outside the module. Modules in which such entries should be used, the name of the modules where the entries are defined must be declared in the access declaration. This rule is for ease of development and management of rather large-scaled software systems.

7.1 Using Entries of Other Modules

An access declaration makes the entries of other modules whose names are given in it accessible from the module. Two modules, "builtin" and "standard" are always accessible by default. As module names are unique in the system, entries of accessible modules can be uniquely denoted using their full names. Access declarations are used by the program database manager. For example, when one module is to be loaded, the loader examines whether modules declared in the access declaration are already loaded. If not, such modules are loaded automatically. Applying this rule recursively, all the required modules can be loaded automatically.

A visibility declaration makes the entries of declared

modules visible from inside the module, i.e., if they have a unique short name among visible entries, they can be denoted by their short names, without prefixing it with the defining module name. Two modules, "builtin" and "standard" are always visible by default. Visibility declarations do not work transitively. When a module is visible and another module is visible from inside that visible module, it does not necessarily mean that the latter is directly visible from the original module.

7.2 Declaring Entries

Entry declarations are used to make entities defined in a module accessible from outside the module. Entities defined in a module but not declared to be entries are accessible only from within the defining module and are said to be local to that module.

In entry declarations, predicates are identified by the principal functor and the arity of their heads. Macros are identified by the principal functor and the arity of their invocation pattern. Global data are identified by their names. Operators are identified by their names and types, i.e., prefix, infix, or postfix. Symbols which are used in the module and not an entry of other visible modules are implicitly defined in the module. They also can be made accessible by their names from outside the module using entry declarations.

Figure 7 shows an example of a module in ESP, which defines a sorter. Here, the predicate concatenate is assumed to be defined in the module list_handler whose name is given in the access and visibility declarations.

8. Conclusion

ESP, along with its object language KL0, has enough description power for development of programs which are readable and also reasonably efficient. There seems to be not much difficulty in implementing the language. We will be using the language in the development of the operating system of our inference machines as well as various other application systems.

Acknowledgments

The design of the KL0 language is a cooperative work of the author, Minoru Yokota and Takashi Hattori of ICOT Research Center. Discussions with the members of the Research Center and working groups of ICOT have been of much help in the design of ESP.

References

1) S.Uchida, M.Yokota, A.Yamamoto, K.Taki and H.Nishikawa: Outline of the Personal Sequential Inference Machine: PSI, New Generation Computing, Vol.1, No.1, Ohmsha and Springer-Verlag (1983).

2) T.Hattori and T.Yokoi: Basic Constructs of the SIM Operating System, New Generation Computing, Vol.1, No.1, Ohmsha and Springer-Verlag (1983).

3) T.Chikayama, T.Hattori, and M.Yokota: A Draft Proposal of Fifth Generation Kernel Language, Version 0.1, Technical Memo TM-007, Institute for New Generation Computer Technology, Tokyo (1982).

4) G.Battani and H.Meloni: Interpreteur du language de programmation PROLOG, Groupe d'Intelligence Artificielle, Marseille-Luminy (1973).

5) R.S.Boyer and J.S.Moore: The Sharing of Structure in Theorem Proving Programs, Machine Intelligence 7, Edinburgh University Press (1972).

6) M.van Caneghem: PROLOG II Manuel D'Utilisation, Groupe d'Intelligence Artificielle, Marseille-Luminy (1982).

```
<operator definition> ::=
    <operator kind> <operators> { <precedence definition> }

<operator kind> ::= "prefix" | "infix" | "postfix"

<precedence definition> ::=
    <precedence relation> [ <direction> ] <operators>

<precedence relation> ::= "precedes" | "is_preceded_by"

<direction> ::= "left" | "right"

<operators> ::=
      <operator>
   | "(" <operator> { "," <operator> } ")"
```

Figure 1. Syntax of Operator Definition

<macro definition> ::=

<invocation pattern> "=>"

<expanded pattern> { <runtime condition> }

":-" <expansion condition>

<invocation pattern> ::= <term>

<expanded pattern> ::= <term>

<runtime condition> ::=

"when" <generator> | "where" <checker>

<generator> ::= <goal list>

<checker> ::= <goal list>

<expansion condition> ::= <body>

Figure 2. Syntax of Macro Definition

Definition:

```
'(X + Y)  =>  Z  when  +(X, Y, Z);
```

Invocation in a Body:

```
... :- ..., p(X + Y), ... ;
==> ... :- ..., +(X, Y, Z), p(Z), ... ;
```

Invocation in a Head:

```
q(X + Y) :- ... ;
==> q(Z) :- ..., +(X, Y, Z);
```

Figure 3. Definition and Usage of "+"

Definition:

```
'(Pattern ! Condition)   =>   Pattern  where  Condition;
```

Usage in a Body:

```
... :- ..., r(X ! X > 0), ... ;
==> ... :- ..., r(X), X > 0, ... ;
```

Usage in a Head:

```
s(X ! X > 0) :- ... ;
==> s(X) :- X > 0, ... ;
```

Figure 4. Definition and Usage of "!"

<global data definition> ::=

<data name> "::" <global data skeleton>

[":-" <initiator>]

<data name> ::= <simple name>

<global data> ::= <term>

<initiator> ::= <body>

Figure 5. Syntax of Global Data Definition

```
<module definition>  ::=
    "module"  <module name>  "is"
        [ <access declaration>  ";" ]
        [ <visibility declaration>  ";" ]
        { <operator definition>  ";" }
        { <entry declaration>  ";" }
        <definition item>  { ";"  <definition item> }  "."

<access declaration>  ::=  "with"  <module name list>

<visibility declaration>  ::=  "use"  <module name list>

<entry declaration>  ::=
      <entry predicate declaration>
    | <entry macro declaration>
    | <entry global data declaration>
    | <entry operator declaration>
    | <entry symbol declaration>
```

Figure 6. Syntax of Module Definition

```
%% Definition of Module "QUICK_SORT"
%%
%%      From "%%" up to the end of line is a comment.

module  quick_sort  is

   %% Access and Visibility Declarations:
   %%
   %%       Use Entries of the Module "list_handler"

    access  list_handler;

    use  list_handler;

   %% Entry Declaration:  Only one entry "sort"

    predicate  sort(Original, Sorted);

   %% Entry Predicate "sort"

    sort([], []);

    sort([X|L], R) :-

        partition(L, X, L1, L2),

        sort(L1, R1),  sort(L2, R2),

        concatenate(R1, R2, R);   %% defined in "list_handler"

   %% Local Predicate "partition"

    partition([X|L], Y, [X|L1], L2)  :-

        X<Y,  partition(L, Y, L1, L2);

    partition([X|L], Y, L1, [X|L2])  :-

        X>=Y,  partition(L, Y, L1, L2);

    partition([], _, [], []).
```

Figure 7. Module Definition Example - Quick-Sort